CS5218 Principal of Program Analysis

# Assignment 3

# Wang Jiadong (A0105703)

Interval analysis is a famous analysis developed in 1950s and 1960s. In this approach, an interval for a variable x is represented by x: [a,b] where a is the lower bound and b is the upper bound on the possible values that x can contain. In this assignment, we will be implementing an Interval analysis with some path sensitivity for all the variables in the input program.

## Task1: Interval Analysis

### Flow (F).

Intuitively, interval analysis is a top-down(forward) analysis. We analyze the range for each variable from init(S\*) to final(S\*), and we adjust the variable value range for each instruction. However, for branching instructions, we do have to do some extra backward analysis, to get the entry value set for its successors.

### Starting Block(E).

Since the analysis flow is top-down, then it is very clearly the starting block is init(S\*).

### Initial Value().

### Since there are no variables declared initially, we just define the initial value as an empty set (Ø). In LLVM, only after some instructions that declare new variables (e.g. Alloca, Load, etc.), shall we add the new variable, along with its interval, into the lattice.

### For instance, for

### **“%1 = alloca i32, align 4”**,

### we add

### **“%1 → [INFINITY\_NEGATIVE, INFINITY\_POSITIVE]”**,

### into the analysis result.

### May or Must.

### We are actually trying to find the “possible” range for each variable. So union function is used, instead of intersection. Hence, it is a “May”.

### Lattice and Partial Order

Here, we use a lattice, with each node as an element the power set of all variables. Hence, each node represents a possible combination of contaminated variables. In this lattice, the bottom is ø (empty set) while the top is the complete set of all variables. Hence, the lattice is partially ordered by subset inclusion: 

### Monotonicity

As mentioned above, the top element is the lattice, is the complete set of all variables, while the bottom element is an empty set. Hence, for any pair of element “a” and element “b” in this lattice, if there is a path from a to b and a <= b, then a is a subset of b.

### Transfer Functions.

Transfer function can be represented as , where:

Kill[ a = exp ] = {a} when a is tainted at the entry while none of the variables inside the exp is tainted.

Gen[a = exp] = {a} when a is not tainted at the entry while one or more of the variables inside the exp is tainted.

When inside the LLVM context:

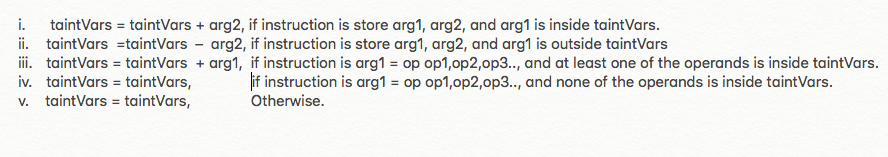


Figure 1 Transfer Function in LLVM.

## Task 2: Implementing Loop-Free Taint Analysis Based On LLVM

### Design.

Task 2 requires us to design a LLVM pass to identify the taint variables for loop free programs. Since it is loop free, we can find the fix point with only one iteration. Hence the idea is that, we find the entry point of the program firstly, and then, block by block, we find the taint variables, feeding them to the successors. This process will terminate at the end point. Since it is an isolated entry & isolated exit program, there should be only one entry point and one exit point.

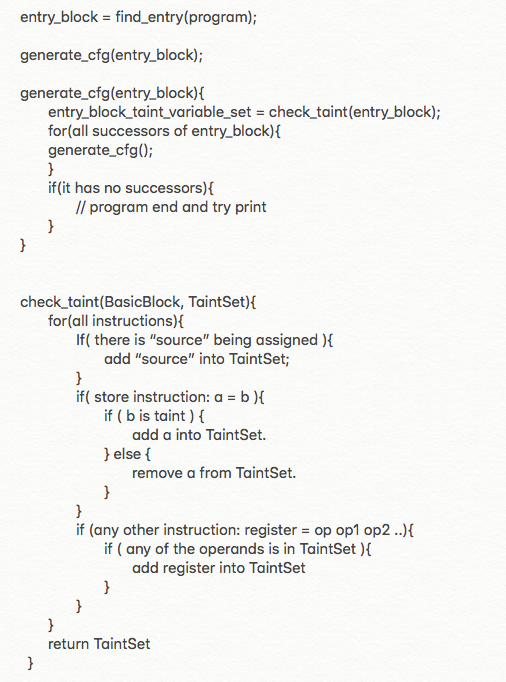


Figure Pseudo-code for loop-free taint detection.

### Building & Run

The code is attached, and building process is straight forward:

This will give us the TaintAnalysis Pass. We can use this pass to conduct taint analysis on testing programs, as:

./TaintAnalysis test1.ll.

### Results

Test Case 1 Source Code:

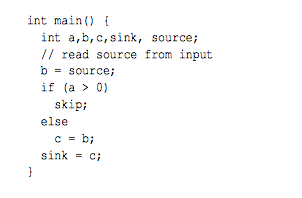


Figure Test Example 1 source code.

Test Case 1 IR Code:

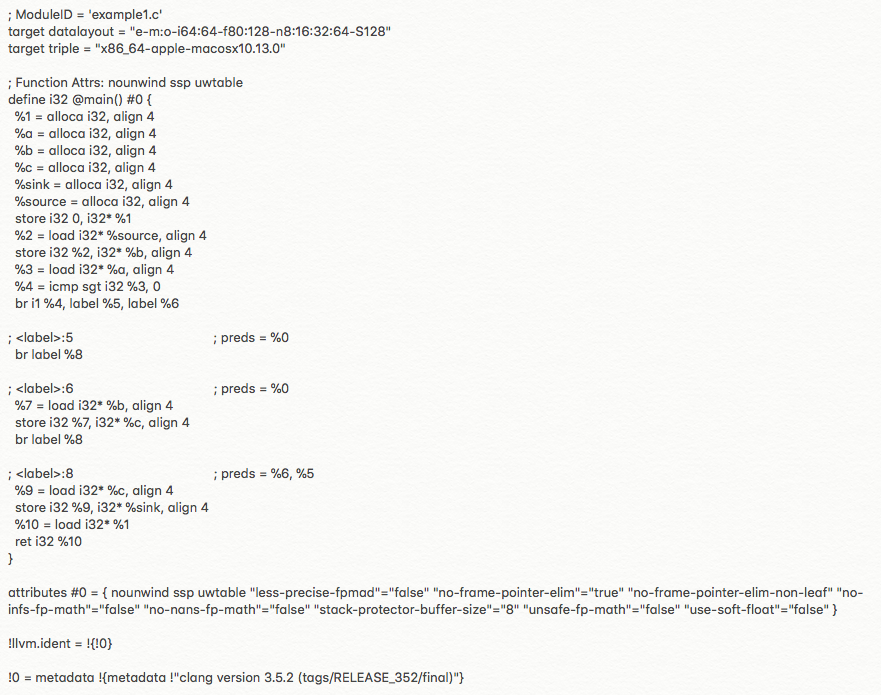


Figure Test Example 1 IR.

Test Case 1 Output:

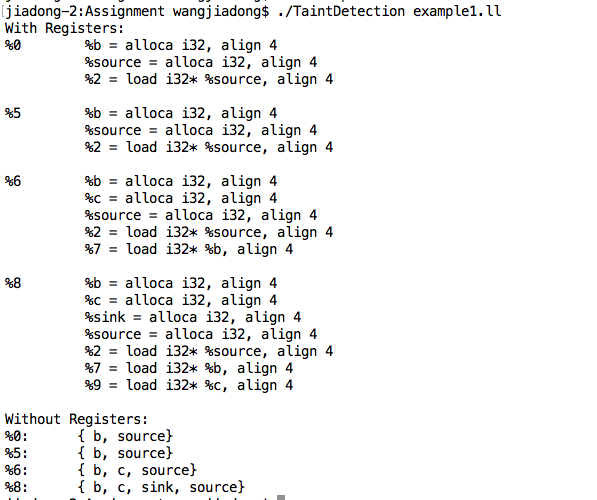


Figure : Test Example 1 Analysis Output.

Test Case 2 Source Code:

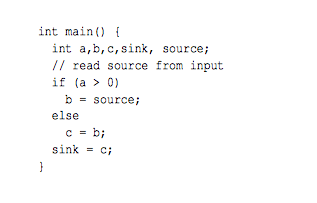


Figure Test Example 2 Source Code.

Test Case 2 IR Source Code:

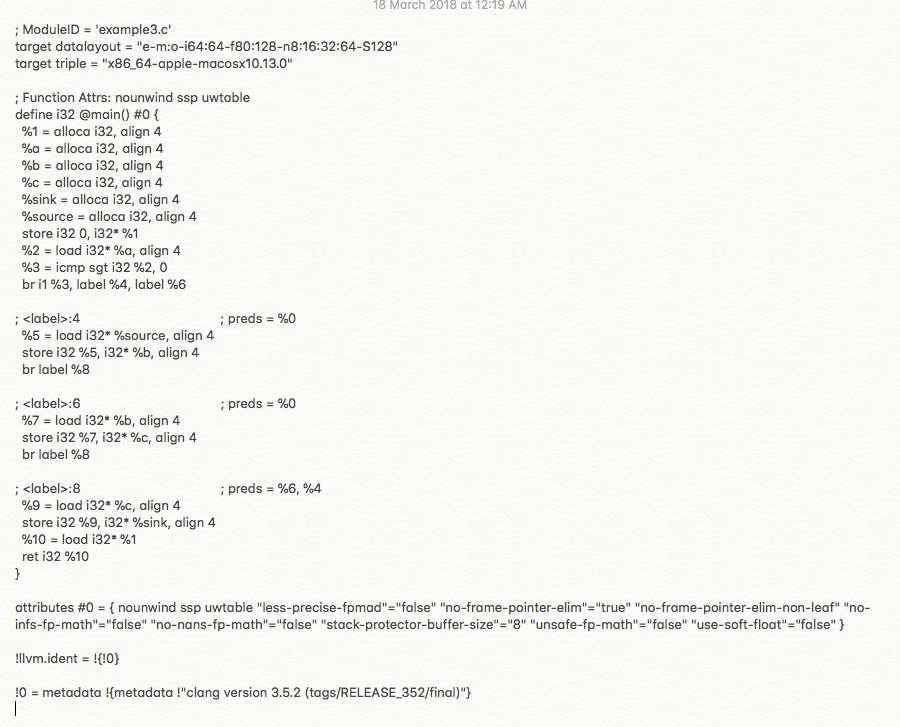


Figure Test Example 2 IR File.

Test Case 2 Output:

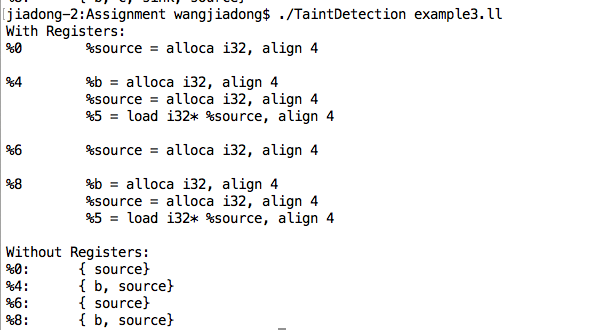


Figure Test Example 2 Output.

## Task 3. Adding Support for Loop in Taint Analysis

### Design

After considering the while loops, the taint analysis becomes more complicated. In this program, we are adopting the depth-first strategy. In order to do so, we use a stack to hold the list of analysis outputs for basic blocks. Every time, we take the top basic block from the stack, and conduct taint analysis. Once completed, we will compare the new outcome with that from last iteration. If the output is updated, which means we have not reached the fix point yet, we will remove this basic block from the stack, and place its successors into it, with the updated entry set, and conduct the analysis consequently. We will reach the fix point once the stack is empty.



Figure With-Loop Taint Analysis.

### Building & Run

The building process is similar with task 2.

### Results

Test Case 3 Source Code:

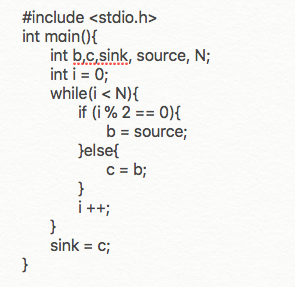


Figure : Test Example 3 Source Code.

Test Case 3 IR Source Code:



Figure Test Example 3 IR File.

Test Case 3 Output I:

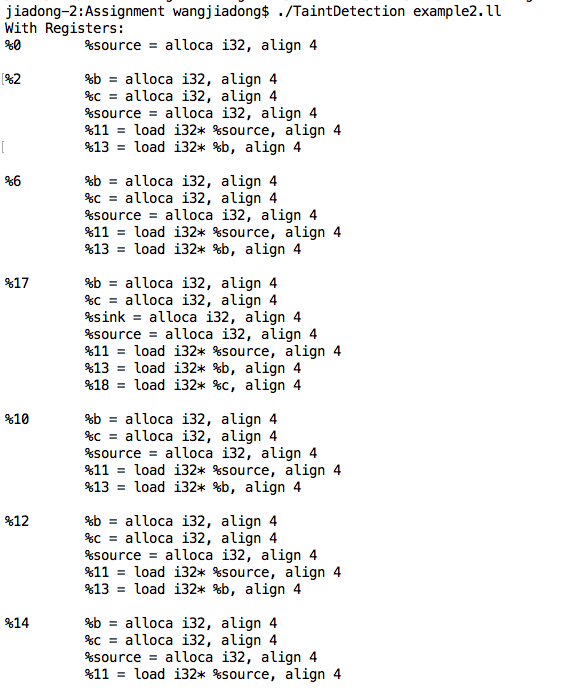


Figure Test Example 3 Output (With Registers).

Test Case 3 Output II:

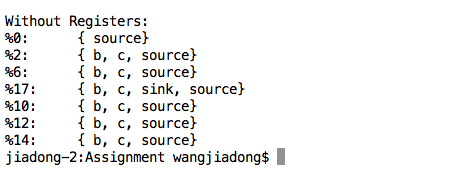


Figure Test Example 3 Output (Without Registers).

Some other test cases are also attached inside the zip file.